A Look Inside the KIPR Link
Braden McDorman, Joshua Southerland
KISS Institute for Practical Robotics, University of Oklahoma
bmcdorman@kipr.org, southou@gmail.com

A Look Inside the KIPR Link

Contents

1 Introduction

The KIPR Link (herein referred to as just "The Link") is the fifth generation educational robotics controller used in the Botball robotics competition. The Link is also the first robotics controller designed from the ground up for Botball. Since the Link runs a full linux kernel and is completely open source, it is a perfect platform for modification and experimentation by more advanced users. This document hopes to serve as a guide for the curious and ambitious user that wishes to modify the Link for their own purposes.

1.1 Disclaimer

Modifying the Link's firmware can lead to system instabilities and possible bricking. While this article does provide documentation for un-bricking any Link manually, these steps require opening the case and **will void your warranty**.

1.2 Document Format

Colored boxes are placed throughout this document to provide caution and insight to the reader:

Hold On: Read everything in red boxes **very carefully**.

Note: Take the following information under advisement.

Why? Provides insight into why certain design decisions were made.

Get It! Provides links to documentation and source files.

Novice's Corner: A high-level overview of a possibly foreign concept, component, or system. These boxes might also contain links to selected readings.

At the end of every section, I will attempt to provide relevant links to documentation, source code, and other useful references for beginning development with the Link.

2 System Overview

2.1 Hardware Overview

While this document is primarily aimed at understanding and modifying the Link's software, a basic understanding of the Link's hardware is essential.

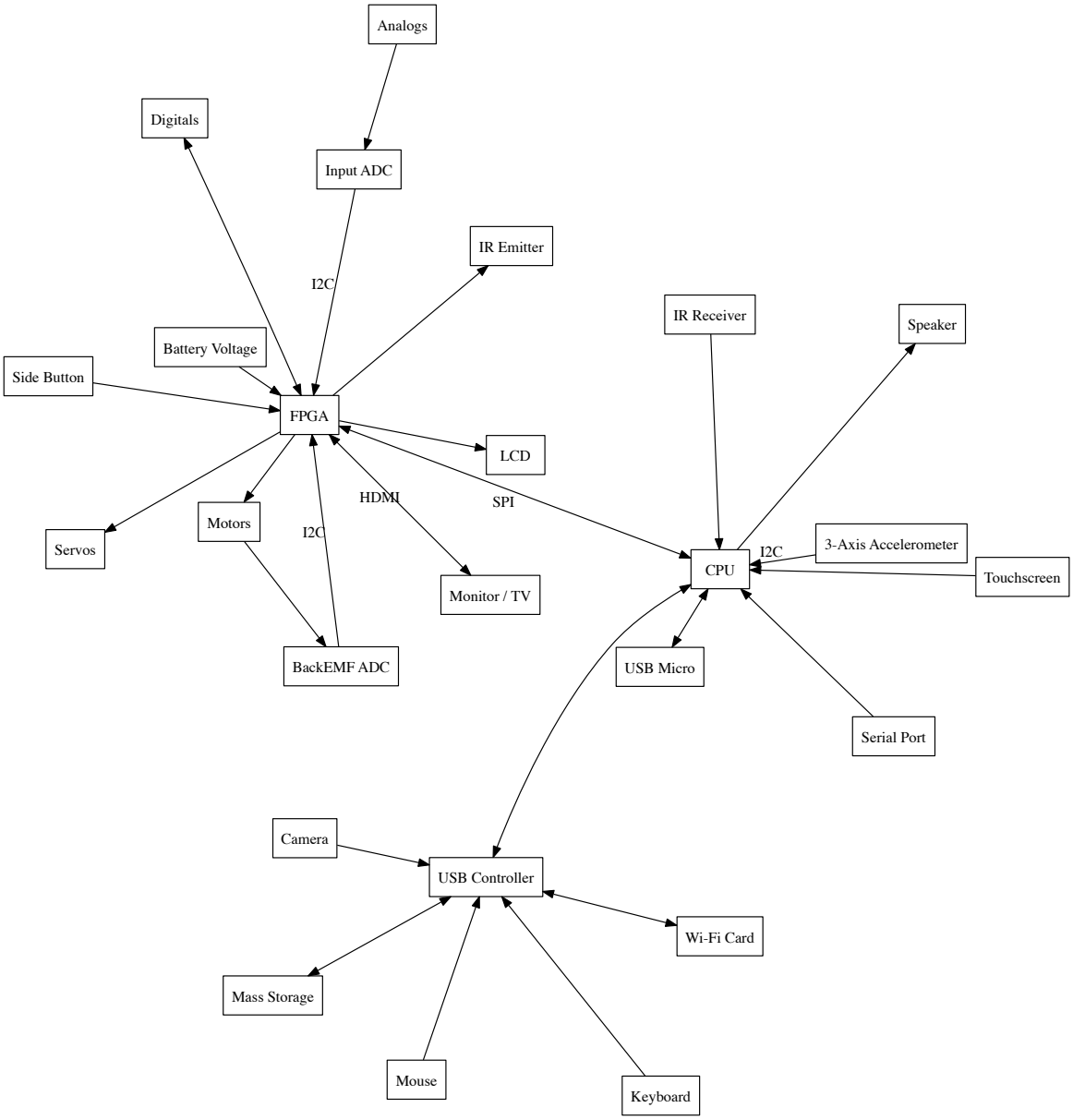

Figure 1: Hardware Overview of the KIPR Link

The Link features an ARMv5te CPU and a Spartan 6 FPGA. Both of these chips work

in tandem to enable all of the functionality of the Link.

Novice's Corner: A FPGA, or Field-Programmable Gate Array, is an integrated circuit that can be programmed to perform highly parallel and fast custom logic. FPGAs can be orders of magnitude faster than CPUs for certain types of operations, but are not designed to handle the sequential logic that a CPU would normally perform.

Unfortunately the FPGA is more of a hindrance than a benefit in the current version of the Link. The FPGA was originally added to mediate fast vision processing like the XBC's. Using a USB camera, however, means that the cost of transporting the image data to the FPGA is more expensive than just doing the computation on the CPU. To realize the performance the XBC enjoyed, the camera would need a direct (and preferably non-USB) connection to the FPGA. **Write More**.

2.2 Software Overview

Most of the software in this document was designed and implemented when the Link was still called by its codename: "Kovan".

Why? Kovan is the name of our hardware designer's bus stop in Singapore. That's all I've got.

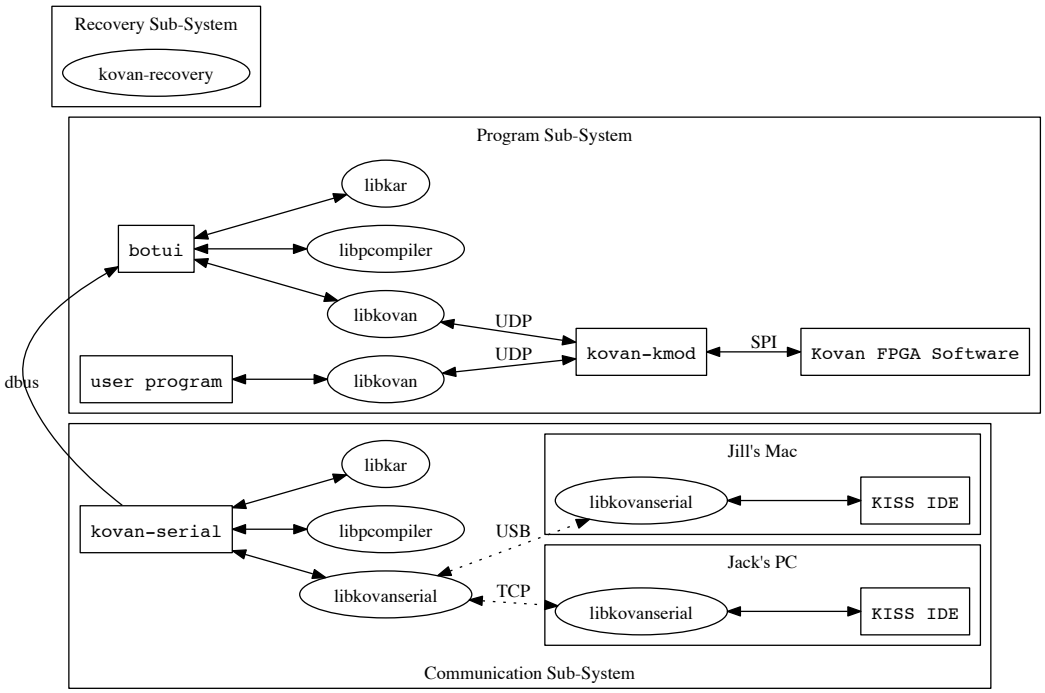

Figure 2: Software Overview of the KIPR Link

Write More.

3 The Software Stack

The Link's full software stack is composed of hundreds of libraries, executables, and configuration files. Fortunately, many of these are not entirely relevant to the discussion of the Link's software components. In this section I will enumerate the important libraries and executables to understand, provide notes on their implementation, provide examples of their usage, and describe how a developer might go about modifying these packages.

3.1 `libkovanserial`

`libkovanserial` is the unified network and USB communication protocol for the Link and KISS IDE. `libkovanserial` is divided into three layers:

1. "Transmitters" are back-ends that implement a specific communication mechanism, such as TCP/IP sockets or USB comm ports. Security is not handled on this layer.
2. The "Transport Layer" handles packet creation, checksumming, and a basic ACK/resend mechanism for non-reliable protocols such as serial communication ports. Session-level security is defined on this layer.

3. The "Protocol Layer" helps facilitate protocol-level communication with the KIPR Link. This is intentionally left as a somewhat leaky abstraction. User password security is implemented on this layer.

3.1.1 Authentication Handshake

`libkovanserial` uses XOR encryption with a mutual shared session key that is negotiated during handshake. XOR encryption is notoriously insecure with plain text, but the use of random data paired with cryptographically secure data makes normal XOR encryption attack vectors useless. Since the session key is a pseudo-randomly generated 512 bits and sessions are short lived, cracking the session key with brute force is unlikely. `libkovanserial` also goes a step further and fills empty space in packets with pseudo-random bytes. This prevents sniffers from detecting the key using zeroed-out sections of packets. This handshake method guards against man-in-the-middle attacks and other sniffers by XOR encrypting the session key during transmission using a private but mutual piece of information: `sha1(password)`. Since the session key is 512 pseudo-random bits and the SHA1 key is a cryptographically strong hash, decoding either piece of information is unlikely. A new session key is generated with every high-level command to the Link, so a session key does not remain valid for more than a few seconds.

3.1.2 Handshake Example

A typical handshake looks like this:

1. Ask the server if it requires authentication. If no, `finish`. If yes, `goto 2`.
2. Send the server our password's MD5 hash.
3. Check if authentication was successful. If no, prompt user for new password and `goto 2`. If yes, decrypt the session key using our password's SHA1 hash and `finish`.

3.2 The `kovan-serial` Daemon

The `kovan-serial` daemon mediates all incoming connections to the Link over USB and Wi-Fi.

3.2.1 Notes on Using the USB Micro Port

The kernel driver used for the USB Micro port on the Link (`otg_serial`) is unfortunately very buggy with the Link's hardware. If the USB cable is ever physically disconnected and then subsequently reconnected to the Link, the kernel driver enters an error state that *can not be directly detected*. Attempting to read or write data while in this error state will eventually lead to the kernel driver locking up, which can not be fixed without power cycling the device. After much experimentation, it was discovered that this error state can be indirectly detected by attempting to `write` an array of size zero to the open USB file descriptor at semi-frequent intervals (`kovan-serial` checks every two seconds). If `write` fails and sets `errno` to `EIO`, close the file descriptor and re-open it. It should be noted that `read` *will not return any*

error other than setting `errno` to `EAGAIN` (An error that means there was no data ready to read, but that the connection is still good), even though attempting to read once the error state is entered could eventually result in the kernel driver locking up.

3.3 libkovan

`libkovan` is the most important library on the Link. It is designed to implement and/or expose every piece of functionality a user would expect from a robotics controller. This includes, but is not limited to:

- Motor actuation
- Servo actuation
- Camera perception
- AR.Drone communication
- iRobot Create communication
- High-level threading routines
- Simple key/value configuration files
- Data collection and export
- Utility functions

Get It! `http://github.com/kipr/libkovan`

3.4 libkar

`libkar` (LIBKissARchive) is an extremely simple Qt-based archive format used to store and transport data in KISS IDE and its targets (including, but not limited to, the Link.) `libkar` is implemented as a flat key/value dictionary with methods to treat the keys as a two-dimensional tree structure when useful. This dictionary is then serialized to

Novice's Corner: *Serialization* is a fancy term for taking a data structure like a color, time, or image and converting it into an array of bytes that can be either stored on a disk or transferred to another computer via a serial connection. Serialization is reversed using *deserialization*.

(needs cleaning up) For more information on how the data is serialized, please see: http://qt-project.org/doc/qt-4.8/datastreamformat.html

Get It! `http://github.com/kipr/libkar`

3.5 pcompiler

pcompiler (short for precedence compiler) is a library that automatically attempts to produce executables from a set of input files. pcompiler uses file suffix precedence to create a weak build ordering, and then applies *transforms* to mutate the input into an output. For example, given the files: main.c, functions.h, and functions.c, pcompiler will generate the following transforms (herein denoted by a →):

1. {functions.h} → ∅ (passthrough transform)
2. {functions.c, main.c} → {functions.c.o, main.c.o} (c transform)
3. {functions.c.o, main.c.o} → {a.out} (o transform)

Novice's Corner: The C language goes through three build phases: preprocess, compile, and link. The preprocess step takes macros like `#include "file.h"` and replaces them with the specified file before passing the .c files to the compiler. The compiler generates .o (*object files)* from the .c files. Object files are linked together to create an executable (the default executable name on linux is a.out)

The passthrough transform is used to retain files in the build directory but remove them from the compilation set. Since .h files are handled by the .c compiler, there is no need for pcompiler to consider them further. pcompiler expects to only produce one output file (or *terminal*), which means keeping the .h around would result in a failed compilation.

Why? While most advanced programmers would prefer to use a Makefile or other build system for their code, the syntax of these files are often esoteric and distract from mastering the basics of programming. Since the KIPR Link's target audience is middle and high school students, the decision was made to avoid manual build systems altogether

Get It! http://github.com/kipr/pcompiler

3.6 botui

botui is the graphical user interface the user is presented upon starting a Link. botui focuses on four primary tasks: Sensor visualization, simple motor/servo control, device configuration, and, perhaps most importantly, managing user programs.

Get It! http://github.com/kipr/botui

3.7 kovan-kmod

kovan-kmod is a kernel module that mediates communication between libkovan instances and the FPGA. The chosen IPC mechanism was UDP, as UDP allows full duplex communication and avoid the synchronization headaches of other IPC mechanisms.

Get It! `http://github.com/kipr/kovan-kmod`

3.8 `kovan-recovery`

Recovery mode is a special boot mode that is used to perform firmware upgrades and recover semi-bricked devices. To understand `kovan-recovery`, it is first necessary to understand the boot process of the Link and the initialization process of linux. The Link uses a custom Master Boot Record (MBR) initially designed by Chumby Industries for the Chumby. (cite) This custom MBR contains two linux kernels: One regular kernel and one special recovery kernel. It is important to reiterate that these kernels *are not* stored on the device's filesystem, but instead directly in the MBR. The process of booting a linux kernel goes something like this:

1. Load the kernel into RAM.
2. The kernel then builds what is called an `initramfs` and executes a process called `init`. `initramfs` contains an extremely minimal filesystem that is used by the `init` process. The idea behind an `initramfs` is that certain operations that don't belong in the kernel can be executed before the main filesystem is loaded. For example, imagine the case in which a filesystem is encrypted. Linux doesn't know how to use an encrypted filesystem, so a developer could design an `initramfs` that loads the correct keys, decrypts the filesystem, and then continues the boot process. It should be noted that the `initramfs` is actually compiled *directly into* the kernel itself.
3. The real root filesystem is loaded and initialization of the system continues.

`kovan-recovery` takes advantage of `initramfs` to "hijack" the boot process before a root filesystem is mounted. This detail is important, as `kovan-recovery` works by overwriting the entire internal SD card. Since the SD card is being overwritten, it can not be used as a root filesystem. Since `kovan-recovery` and linux exist only in RAM, this is not an issue.

As mentioned earlier, the `initramfs` is compiled directly into the kernel. This means that the Link needs two kernels: One special recovery kernel and one regular kernel. Before the linux kernel is loaded, the side button is checked to see if it is pressed down. If it is, the special kernel is loaded instead of the regular one.

`kovan-recovery` itself is an extremely simple C program that uses `zlib` and basic file i/o to inflate and subsequently write to the internal drive. Since including libraries in the `initramfs` is burdensome, `kovan-recovery` performs double-buffered drawing directly to the `/dev/fb` LCD frame-buffer, using an extremely simple bitmapped font to perform font rendering.

Novice's Corner: Unix represents almost every installed device as a special type of file in the `/dev` directory. This means that a developer can open a device and write to it like it was a file. As noted above, `kovan-recovery` opens the LCD as a file and writes pixels at certain offsets that correspond to onscreen (x, y) locations. This concept is applied in several places throughout the system, and is important to understand.

Get It! `http://github.com/kipr/kovan-recovery`

3.9 KISS IDE

An elementary understanding of KISS IDE 4's architecture is important for modifying and extending the Link.

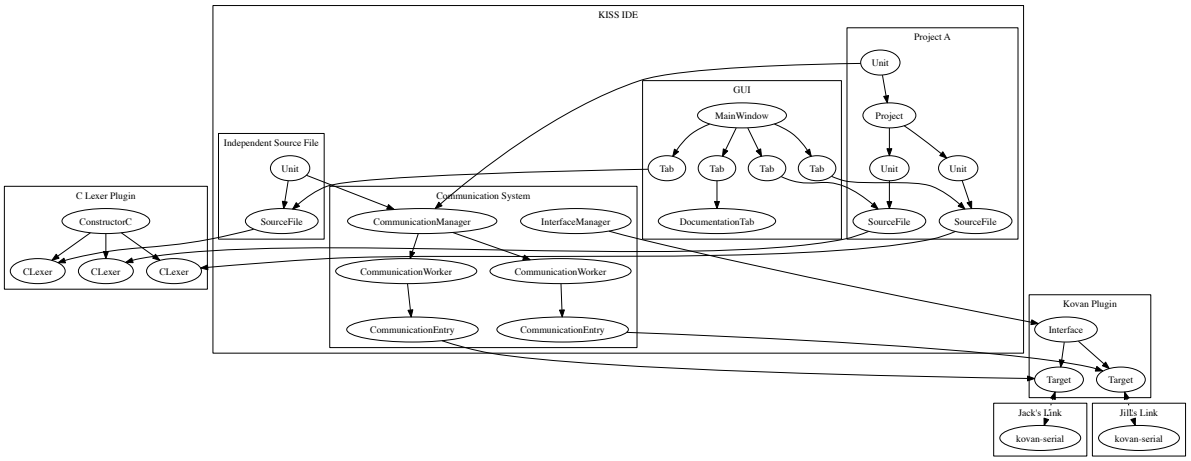

Figure 3: A Hypothetical Runtime Snapshot of KISS IDE

Two of the core concepts in KISS IDE are that of the `Interface` and `Target`. `Interfaces` encapsulate the the discovery, enumeration, and creation of `Targets` for a specific device. A `Target` is used to *download*, *compile*, *run*, and perform other operations on a connected device. Another core concept in KISS IDE is that of the `Unit`. A `Unit` is an abstract class that uses the visitor pattern to create in-memory project and source file archives. For example, imagine the existence of a project called "Task1" that contains three source files: `foo.c`, `bar.h`, and `baz.c`. Now imagine that the user invokes a *download* command on the source file `foo.c`. `foo.c` will ask that its `Unit` invoke a *download* command on the top-level `Unit`, which is Task1. Task1's unit will then create an empty archive, add project files to it, and *visit* its children `Units` recursively. A child `Unit` may then add its required files to the archive. Once this recursive process finishes, the completed archive is sent wrapped in a `CommunicationEntry` and queued in the `CommunicationManager`. The `CommunicationManager` then handles non-blocking execution of the task on the given target.

The most important thing to note here is that KISS IDE *does not* compile source files locally when communicating with a Link. A *compile* command results in the following chain of actions:

1. Download the source archive to the Link
2. Request a remote compilation on the Link.

3. The Link extracts the archive to a temporary directory
4. The Link invokes pcompiler on the directory and blocks until the compilation is completed.
5. KISS IDE then receives these results from the Link.
6. Results are formatted, highlighted, and then presented to the user.

Why? While compile time is negatively influenced by this methodology, there are several notable advantages:

- A firmware exposing new APIs on the Link doesn't require a new version of KISS IDE. This was a large problem with the CBCv2 robotics controller.
- KISS IDE can work without a local compiler.
- The archive format is language neutral, which means languages can be added or removed without changing KISS IDE (although syntax highlighting might not work).

Get It! `http://github.com/kipr/kiss`

4 Building Firmwares and Packages

There is unfortunately no easy way to build firmwares and packages for the Link at this time. This section will, however, describe how to do it the hard way (and the way KIPR does it internally). The Link is based off of a linux distribution system called *OpenEmbedded* and its associated build system called `bitbake`. While these tools have improved dramatically in recent months and years, the Link's build system has not been updated yet. As such, there will be little to no relevant documentation available to assist in the creation of packages or firmwares, and most software available in OpenEmbedded's local repository will be outdated. Fortunately, KIPR has shed most of the tears of frustration on the reader's behalf. What follows is a guide on setting up a build server and modifying pieces of KIPR's software.

4.1 Prerequisites

Note: These instructions require a good amount of experience with linux and distributed version control systems. If you haven't had any experience with those, it is recommended you gain knowledge of them *before* rather than *during* this process.

The following software and hardware is necessary for setting up a Link-compatible OpenEmbedded build server (herein referred to as *the host*):

1. A computer or virtual machine running a *clean install* of the latest Ubuntu Linux. This computer or virtual machine must have:

(a) At least 150 GB of free space (though 300 GB or more is preferred).
(b) At least 4 GB of RAM. KIPR uses 16 GB of RAM.
(c) At least a quad-core CPU. KIPR uses a six-core AMD CPU, but a newer Intel quad-core chip would probably be more performant.
(d) A broadband internet connection.

2. Knowledge of the `git` version control system.
3. Fluent in basic and moderately complex linux terminal commands.

Once Ubuntu Linux has been installed on the host:
`$ sudo apt-get install git subversion binutils`
Insert Clemens' instructions once they are ready

4.2 Core Concepts in OpenEmbedded and `bitbake`

OpenEmbedded organizes build *recipes* into "meta" *layers*. **Write More**.

4.3 Modifying an Existing Package

First, change the working directory to `oe` using `$ cd oe`. Once inside the `oe` directory, source in the build environment using the command `$ . ~/.oe/environment-oecore`. This step must be done every time a new terminal is used.

To tell bitbake to pull new code from a given package's repository, the PR version of the package must be incremented. For example, to increment the PR version of botui using vim, one could do the following:
`$ vim sources/meta-kipr/recipes-kovan/botui/botui_git.bb`
Then enter the characters: `?`, `P`, `R`, `[Enter]`, `Ctrl-A`, `:`, `w`, `q`, `[Enter]`.

Novice's Corner: Vim is an advanced text editor that can be used to modify files in a terminal. The first part of this command is `?PR[Enter]`. This tells vim to find the first occurrence of the string "PR" in the text file and move the cursor to it. `Ctrl-A` tells vim to find the next integer (in this case the PR number) and increment it by 1. Finally, `:wq[Enter]` tells vim to write its changes to disk and then quit.

Now, rebuild the package using `$ bitbake botui`.

If this command completes successfully or with only warnings, congratulations! An installable package file was created and deposited in the `build/tmp-*/deploy/ipk/armv5te` directory! For installation instructions, see the section documenting `opkg`.

4.4 A Custom Package

4.5 Debugging Bitbake Errors

Bitbake will sometimes simply fail with a message containing the word "ERROR". Since there are thousands of things "ERROR" could mean, this will often leave the developer

scratching their heads in frustration. I have discovered, however, that most of these "ERROR" messages with zero information are often the result of a syntax error in a `.bb` package file. As such, it is strongly recommended that developers incrementally test changes to their `.bb` files. If that doesn't help, the developer will need to go through their recently modified `.bb` files and temporally remove them from the repository to check if that file is indeed the problem. Once the problematic file has been identified, comment out portions of the file to determine wherein the error lies.

4.6 Installing Packages and Building Firmwares

OpenEmbedded will automatically generate package files that are installable on the Link using a flash drive and terminal. **Write More**.

4.7 Developing Without a Build Server

Several of the aforementioned software packages will compile and run on a regular computer. **Write More**.

5 Essential Tools for Working with the Link

Write More.

5.1 Secure Shell (SSH)

Note: As of firmware 1.9.8, `botui` sets the root's password to the password configured in Home→Settings→Communication.

`ssh` allows root-level access to the Link over a Wi-Fi connection and is often the most convenient method of modifying the device. First connect the Link to a Wi-Fi network. **Write More**.

5.2 `opkg` Package Manager

The Link's default package manager is called `opkg`. `opkg` can be used to install packages built using a Link build server. **Write More**.

6 Modification Checklists

When creating a component for the Link, there are often several steps that are often obscure and non-obvious. These checklists hope create a discrete list of steps for common Link modifications a developer might want to perform.

6.1 Adding Support For a New Language

- ☐ Write the required `bitbake` *recipes* to create binary packages for the language's compiler and libraries.
- ☐ Write the necessary `pcompiler` *transforms* to compile the language into a binary.
- ☐ Add the language's file extension to `botui`'s `FileActionCompile` class to enable compiling from the Link's embedded file manager.
- ☐ Write a `Lexer` (syntax highlighter) plugin for KISS IDE.
- ☐ Create a *Template Pack* for KISS IDE (File→New→New Template Pack...).
- ☐ Add the required language packages to `meta-kipr/recipes-kovan/images/kovan-kipr-image.bb` to make the language a part of the official firmware.

6.2

- ☐ Write the required `bitbake` *recipes* to create binary packages for the language's compiler and libraries.
- ☐ Write the necessary `pcompiler` *transforms* to compile the language into a binary.
- ☐ Write a `Lexer` (syntax highlighter) plugin for KISS IDE.
- ☐ Create a *Template Pack* for KISS IDE (File→New→New Template Pack...).

7 Conclusion

While this document covers several pieces of the Link's firmware and attempts to give context to certain features, it is by no means comprehensive. If you have any questions or concerns, please feel free to contact the authors for more information. **Write More**.

7.1 Acknowledgements

- Thank you to Nafis Zaman for his contributions to `botui` and KISS IDE.
- Thank you to Clemens Koza for his efforts in creating reliable instructions for setting up a build server from scratch.
- Thank you to the users that have coped with the early instabilities of the new system.

8 Appendix

8.1 Manually Un-bricking a Link

These steps will void your warranty.
If the Link fails to boot to standard mode or recovery mode, these steps can be used to manually flash the internal micro-SD card. The following steps require a Torx screwdriver, a micro-SD card reader (or SD card reader and adapter), and an Operating System supporting `dd` (*nix or Mac OS X).

1. Remove the warranty sticker from the Link's case.
2. Remove the four bottom screws using a Torx screwdriver.
3. Locate the micro-SD card hinge (by the power switch).
4. Push the hinge towards the inside of the case and lift up.
5. Place the recovered micro-SD card in card reader.
6. The OS will probably auto-mount the contents of the drive and possibly display some error messages. Ignore these error messages and unmount the SD card if auto-mounted.
7. Determine the appropriate device file to write to in `/dev`. This can easily be determined by removing the card, running `$ ls /dev > 1`, inserting the card again, and running `$ ls /dev > 2 && diff 1 2`. The correct device file will not have any numerical suffix.
8. Download the KIPR Link firmware file to install.
9. Extract the KIPR Link firmware file using `gunzip`.

Hold On: Triple check that the determined device file is indeed the micro-SD card. Choosing the wrong device file could overwrite your hard drive disk!

Finally, `$ sudo dd if=firmware_file.img of=/dev/determined_device_file bs=1M` (the M might be lowercase on your platform).

This document was compiled on June 16, 2013.